Fonctions élémentaires en virgule flottante pour les accélérateurs reconfigurables

Jérémie Detrey, Florent De Dinechin

Fonctions élémentaires en virgule flottante pour les accélérateurs reconfigurables

Jérémie Detrey — **Florent de Dinechin**

Laboratoire de l'Informatique du Parallélisme
Ecole Normale Supérieure de Lyon
46, allée d'Italie
69 364 Lyon cedex 07

{Jeremie.Detrey,Florent.de.Dinechin}@ens-lyon.fr

RÉSUMÉ. *Les circuits reconfigurables FPGA ont désormais une capacité telle qu'ils peuvent être utilisés à des tâches d'accélération de calcul en virgule flottante. La littérature (et depuis peu les constructeurs) proposent des opérateurs pour les quatre opérations. L'étape suivante est de proposer des opérateurs pour les fonctions élémentaires les plus utilisées. Parmi celles-ci, nous proposons des architectures dédiées pour l'évaluation des fonctions exponentielles, logarithme, sinus et cosinus, et étudions les compromis possibles. Pour chacune de ces fonctions, un seul de ces opérateurs surpasse d'un facteur dix les processeurs généralistes en terme de débit, tout en occupant une fraction des ressources matérielles du FPGA. Tous ces opérateurs sont disponibles librement sur* `http://www.ens-lyon.fr/LIP/Arenaire/`.

ABSTRACT. *The capacity of reconfigurable circuits nowadays allows them to tackle floating-point acceleration tasks. Operators for the basic operations have been described in the literature and are now available from FPGA vendors. The next step is to provide operators for the main elementary functions. This article presents operator architectures for exponential, logarithm, sine and cosine, and studies the tradeoffs involved. For each of these functions, one of these operators has ten times the throughput of a general-purpose processor, while consuming a small fraction of the FPGA resources. These operators are freely available at* `http://www.ens-lyon.fr/LIP/Arenaire/`.

MOTS-CLÉS : *Virgule flottante, sinus, cosinus, exponentielle, logarithme, opérateurs matériels, FPGA, VHDL.*

KEYWORDS: *Floating point, sine, cosine, exponential, logarithm, hardware operators, FPGA, VHDL.*

1. Introduction

1.1. *Virgule flottante sur FPGA*

Longtemps cantonnés au domaine de la virgule fixe, les circuits reconfigurables sont depuis les années 2000 de plus en plus utilisés pour implémenter des applications de calcul en virgule flottante. La virgule flottante permet de manipuler des nombres réels de grande dynamique, et est à ce titre d'un emploi plus simple que la virgule fixe. En particulier, elle est utilisée pour développer et tester en logiciel des algorithmes numériques complexes qu'on aimerait pouvoir ensuite implanter rapidement sur FPGA.

Toutefois, les opérateurs flottants pour les quatre opérations sont plus complexes que les opérateurs en virgule fixe. Implémentés sur FPGA, ils payent le prix de la reconfigurabilité et sont beaucoup plus lents que les opérateurs flottants très optimisés des processeurs du commerce. Typiquement, un opérateur flottant sur FPGA a un débit de crête (en termes d'opérations par seconde) au moins dix fois inférieur à son équivalent dans un processeur de la même génération (Ligon *et al.*, 1998 ; Underwood, 2004).

Pour surpasser en performance le processeur, une application flottante sur FPGA doit donc :

– soit présenter un parallélisme massif (Lienhart *et al.*, 2002 ; Govindu *et al.*, 2004 ; deLorimier *et al.*, 2005 ; Dou *et al.*, 2005), auquel cas on cherchera à optimiser le nombre d'opérateurs flottants par circuit FPGA en utilisant une précision *ad hoc* pour l'application (Detrey *et al.*, 2005a), là où le processeur ne laisse le choix qu'entre double et simple précision ;

– soit nécessiter des calculs pour lesquels les opérateurs matériels du processeur ne sont pas optimisés. C'est le cas des opérateurs pour les fonctions élémentaires présentés ici.

1.2. *Le calcul des fonctions élémentaires en virgule flottante*

Faut-il consacrer du silicium des processeurs à des unités dédiées au calcul de fonctions élémentaires ? Si cette question a fait débat (Paul *et al.*, 1976), le consensus actuel est que ce silicium sera mieux employé à améliorer les unités flottantes de base et augmenter leur nombre. Les fonctions élementaires sont donc essentiellement implémentées en logiciel (Tang, 1991 ; Markstein, 2000 ; Muller, 2005), et ce même pour les processeurs compatibles x86 qui, par héritage, offrent des instructions de calcul des principales fonctions élémentaires (Anderson *et al.*, 2006).

La question se pose différemment dès lors que l'on cible un FPGA : un opérateur matériel pour une fonction élémentaire n'y occupera en effet des ressources que dans une application qui en a effectivement besoin. De plus, il sera possible de le tailler au plus juste pour l'application. Le présent article, qui reprend et étend des publications précédentes (Detrey *et al.*, 2005c ; Detrey *et al.*, 2005b ; Detrey *et al.*, 2006),

montre que cette flexibilité du FPGA lui permet alors de surpasser en débit l'implémentation dans le processeur. Pour la simple précision, on peut ainsi obtenir un débit dix fois plus élevé, là où les opérateurs de base avaient un débit dix fois plus faible. On obtient de telles performances par l'utilisation d'algorithmes matériels spécifiques, dont trois exemples sont donnés pour le logarithme en section 2, pour l'exponentielle en section 3, et pour les fonctions sinus et cosinus en section 4. Ces implémentations sont compatibles avec les fonctions de la bibliothèque mathématique standard (ISO/IEC, 1999), ce qui permet de transposer rapidement un code C ou Matlab sur FPGA sans crainte d'obtenir un comportement numérique différent.

Mais nous proposons aussi, en section 5, différentes implémentations de fonctions trigonométriques non standard. En effet, la spécification standard, qui définit un argument en radian, est coûteuse à implémenter précisément, que ce soit en logiciel ou en matériel. Plus précisément, c'est la réduction d'un nombre flottant arbitraire dans l'intervalle $[-\pi, \pi]$ qui est coûteuse du fait de l'irrationnalité de π. Or on observe que de nombreux codes scientifiques ou de traitement du signal passent au sinus un argument qui a été multiplié préalablement par un terme en π. Cette multiplication, coûteuse elle-même, rend aussi bien plus coûteuse l'implémentation des fonctions trigonométriques utilisées derrière[1]. Un autre cas particulier courant est celui où l'argument d'entrée est toujours compris entre $-\pi$ et $+\pi$. En logiciel, d'ailleurs, ce cas particulier est traité bien plus rapidement que le cas général. Nous proposons un opérateur optimisé aussi pour ce cas.

Ce travail est à notre connaissance inédit. Le seul article comparable de la littérature (Ortiz *et al.*, 2003) est extrêmement flou quant à la spécification de la fonction effectivement implémentée.

La section 6 donne des résultats de synthèse pour ces opérateurs jusqu'à la simple précision, ainsi qu'une comparaison avec les performances d'un processeur Pentium Xeon à 2,4 MHz. La section 7 donne quelques pistes pour atteindre la double précision.

1.3. *Contributions : précision et performance*

Dans la littérature, seuls deux articles abordent le sujet des fonctions élémentaires sur FPGA, pour les fonctions sinus (Ortiz *et al.*, 2003) et exponentielle (Doss *et al.*, 2004). Dans les deux cas, les algorithmes utilisés restent très proches des algorithmes logiciels, et en particulier sont exprimés en termes d'opérations flottantes.

Plutôt que d'effectuer tous les calculs intermédiaires en virgule flottante, que l'on sait moins performante que la virgule fixe, notre approche est se débarrasser au plus

1. La révision de la norme définissant le calcul flottant (voir `http://754r.ucbtest.org/`) devrait préconiser de fournir également $sin(\pi x)$, mais il faudra de nombreuses années avant que ces fonctions se généralisent.

tôt de ce format de représentation pour n'avoir plus que des calculs en virgule fixe jusqu'à la reconstruction finale du résultat.

Une autre originalité de nos travaux est l'utilisation de grosses tables de valeurs précalculées pour évaluer des fonctions en virgule fixe. Ces tables sont compressées par la méthode HOTBM (Detrey *et al.*, 2005d ; Detrey, 2007). Sans rentrer dans les détails, cette méthode remplace une grande table contenant les valeurs d'une fonction continue par un opérateur d'interpolation à base de tables plus petites, d'additions et de petites multiplications.

D'après le dilemme du fabriquant de tables (Lefèvre *et al.*, 1998), garantir l'arrondi correct du résultat demanderait de calculer la fonction avec une précision bien supérieure à la précision cible. Nous nous contentons, comme dans les bibliothèques logicielles usuelles, d'assurer l'arrondi fidèle, c'est-à-dire une erreur absolue d'au plus un bit sur la mantisse du résultat. Pour obtenir cette précision, une analyse d'erreur détaillée est réalisée pour chaque opérateur. Cette analyse d'erreur permet également d'optimiser finement les chemins de données.

Enfin, tous les opérateurs présentés ont été testés exhaustivement pour les différents couples de paramètres (w_E, w_F) jusqu'à la simple précision $(w_E, w_F) = (8, 23)$ sur une carte Celoxica RC1000-PP couplée à un PC. Ces simulations ont ainsi permis de vérifier la validité de notre analyse d'erreur : tous les résultats sont bien un arrondi fidèle de la valeur théorique, et un arrondi correct dans plus de 75 % des cas.

1.4. *Notations*

Nous utilisons dans tout l'article les notations de la bibliothèque FPLibrary (Detrey *et al.*, 2005a), librement accessible depuis `http://www.ens-lyon.fr/LIP/Arenaire/`, et dans laquelle s'insère ce travail.

La dynamique des nombres manipulés est donnée par w_E, le nombre de bits de l'exposant (E_x), et leur précision est donnée par w_F, le nombre de bits de la partie fractionnaire de la mantisse (F_x). Pour mémoire, en simple précision, on a $w_E = 8$ et $w_F = 23$. Les nombres flottants comptent aussi un bit de signe (S_x), ainsi que deux bits supplémentaires (exn_x) permettant de gérer simplement les cas exceptionnels tels que zéros, infinis et NaN (*Not a Number*).

La taille d'un nombre flottant dans ce format est donc de $w_E + w_F + 3$ bits, comme représenté en figure 1. Ce nombre x a pour valeur

$$x = (-1)^{S_x} \times \overline{1, F_x} \times 2^{E_x - E_0}, \quad \text{avec} \quad E_0 = 2^{wE-1} - 1.$$

Ici, $\overline{1, F_x}$ dénote le nombre rationnel dont l'écriture en binaire est $1, F_x$. Cette convention sera utilisée tout au long de cet article.

Une autre convention sera de noter les valeurs réelles infiniment précises en minuscule, et les approximations manipulées dans les architectures en majuscule. Ainsi, la contrainte de l'arrondi fidèle s'exprime comme suit :

$$\frac{|r-R|}{2^{E_R-E_0}} = \left| \frac{|r|}{2^{E_R-E_0}} - \overline{1,F_R} \right| < 2^{-w_F}.$$

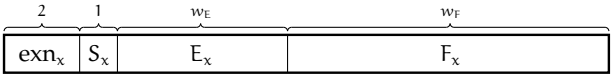


Figure 1. *Format d'un nombre flottant x*

2. La fonction logarithme

2.1. *Propriétés et réduction d'argument*

On se place ici dans le cas où X est un nombre strictement positif représenté en virgule flottante. Dans le cas contraire, l'opérateur doit retourner NaN. On a ainsi :

$$X = \overline{1,F_X} \cdot 2^{E_X-E_0}.$$

On notant $r = \log X$, on obtient :

$$r = \log\left(\overline{1,F_X}\right) + (E_X - E_0) \cdot \log 2.$$

Dans ce cas, il suffit alors de calculer la valeur de $\log\left(\overline{1,F_X}\right)$ avec $1 \leq \overline{1,F_X} < 2$ et d'y ajouter le produit $(E_X - E_0) \cdot \log 2$ pour obtenir le résultat final.

Cependant, lorsque X est proche de 1 par valeurs inférieures, on a $E_X - E_0 = -1$ et $\overline{1,F_X}$ proche de 2, ce qui nous donne :

$$r = \log\left(\overline{1,F_X}\right) + (E_X - E_0) \cdot \log 2 \approx \log 2 - \log 2.$$

L'addition finale va alors provoquer une annulation catastrophique des bits de poids fort (cancellation) et donc une grande perte de précision sur le résultat. Pour éviter cela, on va centrer l'intervalle de sortie du calcul de $\log\left(1,F_X\right)$ autour de 0 : on calcule le logarithme comme

$$r = \log M + E \cdot \log 2,$$

avec :

$$\begin{cases} M = \overline{1,F_X} \text{ et } E = E_X - E_0 & \text{lorsque } \overline{1,F_X} \in [1, \sqrt{2}[, \text{ et} \\ M = \overline{0,1F_X} \text{ et } E = E_X - E_0 + 1 & \text{lorsque } \overline{1,F_X} \in [\sqrt{2}, 2[. \end{cases}$$

De cette manière, on a $\frac{\sqrt{2}}{2} \leq M < \sqrt{2}$, ce qui donne :

$$-\frac{1}{2}\log 2 \leq \log M < \frac{1}{2}\log 2.$$

Dans l'architecture, on est obligé d'approcher la valeur de $\sqrt{2}$ utilisée pour séparer les deux cas ci-avant. Cela n'est pas gênant, et l'approximation de $\sqrt{2}$ utilisée peut même être très imprécise : l'intervalle de $\log M$ est alors plus grand, ce qui peut signifer que son évaluation sera plus chère, mais l'identité $r = \log M + E \cdot \log 2$ n'en restera pas moins vraie. A l'extrême, on peut approcher $\sqrt{2}$ par $1,5$, ce qui revient à implanter la comparaison à $\sqrt{2}$ par le test du premier bit de la mantisse.

2.2. *Evaluation à base de table du logarithme de la mantisse*

Il faut à présent évaluer $\log M$ avec une précision suffisante pour pouvoir garantir l'arrondi fidèle, même après une possible renormalisation du résultat. Or, $\log M$ peut être très proche de 0 au voisinage de 1 : son développement de Taylor est $Log(M) \approx (M-1) - (M-1)^2/2$. Cela montre qu'une évaluation directe de $\log M$ demanderait une précision d'au moins $2w_F$ bits, suivie d'une renormalisation pouvant faire perdre jusqu'à w_F bits.

Ce développement de Taylor suggère une solution plus économe : on va d'abord évaluer $\frac{\log M}{M-1}$ avec une précision de $w_F + g_0$ bits (g_0 représentant un certain nombre de bits de garde défini plus loin), puis multiplier cette valeur par $M - 1$ (calculé exactement).

L'évaluation en virgule fixe de la fonction $\frac{\log M}{M-1}$ est réalisée grâce à la méthode HOTBM (Detrey *et al.*, 2005d ; Detrey, 2007).

2.3. *Architecture complète du logarithme*

L'architecture obtenue est décrite par la figure 2. Quelques remarques et commentaires sur cette architecture :

– le signe de r (et donc de R) est directement obtenu à partir du signe de $E_X - E_0$;

– la précision des calculs intermédiaires est définie par des nombres de bits de garde g_0 et g_1 qui seront déterminés par l'analyse d'erreur ;

– l'addition de $E \log 2$ et de $\log(M)$ est une addition en virgule fixe qui donne un résultat Y. Ce nombre doit ensuite être converti en un nombre flottant. Pour cela on utilise un compteur de zéros de poids fort *(leading zero counter)*. Ce compte définit l'exposant du résultat ($E_R - E_0 = \lfloor \log_2 Y \rfloor$), et est utilisé pour décaler Y pour obtenir un nombre Z normalisé ;

– ce décalage peut être d'au plus w_F bits vers la gauche ou $w_E - 2$ bits vers la droite. En effet, la plus petite valeur non nulle de Y sera obtenue pour $X = 1 + 2^{-w_F}$ (qui donnera $Y \approx 2^{-w_F}$) et la plus grande pour $X = (2 - 2^{-w_F}) \cdot 2^{E_0}$ (soit $Y \approx (E_0 + 1) \cdot \log 2 = 2^{w_E - 1} \cdot \log 2$) ;

– après cette normalisation, il faut encore arrondir Z, qui a été calculé avec des bits de garde, à la précision w_F ;

– enfin, les éventuels cas de dépassement de capacité vers 0 (*underflow*) sont gérés par l'unité de gestion des exceptions.

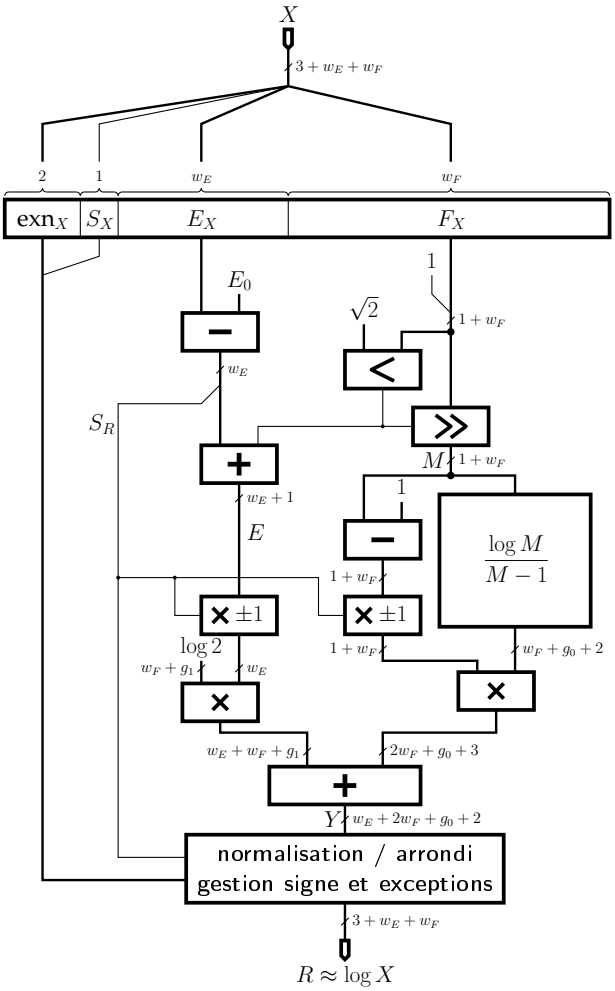


Figure 2. *Architecture du logarithme*

2.4. *Analyse d'erreur*

Y est donc successivement normalisé en Z puis arrondi. Le nombre Z vérifie :

$$1 \leq Z = Y \cdot 2^{-\lfloor \log_2 Y \rfloor} < 2.$$

Pour obtenir la mantisse de R, Z est ensuite arrondi au plus proche à w_F bits de partie fractionnaire, ce qui induit une erreur additionnelle d'au plus 2^{-w_F-1}.

La contrainte de l'arrondi fidèle nous donne donc pour Z :

$$\left| \frac{|r|}{2^{E_R - E_0}} - Z \right| < 2^{-w_F - 1},$$

d'où l'on déduit alors la contrainte de précision sur Y :

$$\left| \frac{|r|}{2^{E_R - E_0}} - Y \cdot 2^{-\lfloor \log_2 Y \rfloor} \right| = \frac{||r| - Y|}{2^{\lfloor \log_2 Y \rfloor}} < 2^{-w_F - 1},$$

soit enfin :

$$||r| - Y| < 2^{-w_F - 1} \cdot 2^{\lfloor \log_2 Y \rfloor}.$$

Il nous faut alors étudier différents cas selon la valeur de E :

– lorsque $|E| > 3$, on a $Y > 2$, et l'erreur prédominante est due à l'erreur de discrétisation commise sur la constante $\log 2$, arrondie à $w_F + g_1$ bits de partie fractionnaire. Cette erreur est ensuite multipliée par $\pm E$;

– à l'inverse, lorsque $E = 0$, la seule erreur commise est l'erreur d'évaluation de $\frac{\log M}{M-1}$, qui est ensuite multipliée par $\pm(M-1)$;

– lorsque $|E| = 2$ ou 3, à la fois l'erreur sur $\log 2$ et sur $\frac{\log M}{M-1}$ doivent être prises en compte. Cependant, dans ce cas-là, on a toujours $Y > 1$, ce qui signifie que la renormalisation de Y n'amplifiera pas cette erreur en la décalant vers la gauche ;

– enfin, lorsque $|E| = 1$, l'erreur est aussi due à la fois à la discrétisation de $\log 2$ et à l'évaluation de $\frac{\log M}{M-1}$, mais on a cette fois-ci l'encadrement suivant sur Y :

$$0{,}34 < \frac{1}{2} \log 2 \leq Y \leq \frac{3}{2} \log 2 < 1{,}04.$$

Dans ce cas, la renormalisation entraînera un décalage d'au plus deux bits vers la gauche, ce qui correspond à une multiplication de l'erreur par un facteur au plus 4.

Ainsi, en calculant l'erreur commise dans chacun de ces quatre cas, on peut facilement en déduire les valeurs minimales des nombres de bits de garde g_0 et g_1. On trouve alors $g_1 = 3$ bits ainsi que la contrainte que l'erreur commise par l'opérateur HOTBM calculant $\frac{\log M}{M-1}$ soit inférieure à $2^{-w_F - 3}$, ce qui nous donne g_0 généralement compris entre 2 et 5 bits.

3. La fonction exponentielle

3.1. *Propriétés et réduction d'argument*

La fonction exponentielle est définie sur l'ensemble des nombres réels, mais les nombres représentables en virgule flottante sont encadrés en valeur absolue par :

$$X_{\min} = 2^{1 - E_0} \leq X \leq X_{\max}(2 - 2^{-w_F}) \cdot 2^{E_0}.$$

Ainsi, la fonction exponentielle doit renvoyer 0 (*underflow*) pour tous les opérandes plus petits que $\log X_{\min} = (1 - E_0) \cdot \log 2 = (2 - 2^{w_E - 1}) \cdot \log 2$, et $+\infty$ (*overflow*) pour tous ceux supérieurs à $\log X_{\max} < (1 + E_0) \cdot \log 2 = 2^{w_E - 1} \cdot \log 2$. Nous nous restreindrons donc dans la suite à des opérandes compris dans l'intervalle $\left[(2 - 2^{w_E - 1}) \cdot \log 2, 2^{w_E - 1} \cdot \log 2\right[$.

3.2. *Algorithme naïf*

On peut calculer l'exponentielle r d'un nombre X en virgule flottante en utilisant l'égalité suivante :

$$r = e^X = 2^{\left(\frac{X}{\log 2}\right)}.$$

Il suffit alors de calculer en virgule fixe le produit de X par $\frac{1}{\log 2}$, noté $Y = \overline{I_Y, F_Y}$. La partie entière I_Y correspondra donc à l'exposant $E_R - E_0$ de la représentation en virgule flottante du résultat R, et $2^{\overline{0,F_Y}}$ nous donnera sa mantisse $\overline{1,F_R}$.

Cependant, cette méthode pose plusieurs problèmes :

– le produit de X par $\frac{1}{\log 2}$ doit être calculé avec une grande précision pour garantir la précision du résultat. Une analyse d'erreur rapide montre qu'il faut au moins $w_E + w_F$ bits de partie fractionnaire pour la constante $\frac{1}{\log 2}$, ce qui implique un très gros multiplieur pour le calcul du produit par X ;

– l'évaluation précise de $2^{\overline{0,F_Y}}$ sera elle aussi coûteuse puisqu'il lui faut w_F bits en entrée.

3.3. *Algorithme amélioré*

Un algorithme un peu plus complexe, plus proche de ce qui est généralement fait en logiciel, permet de résoudre les deux problèmes précédents. L'idée principale ici est de réduire l'argument X à un entier k et un nombre en virgule fixe $Y \in \left]-\frac{1}{2}, \frac{1}{2}\right[$ tels que :

$$X \approx k \cdot \log 2 + Y,$$

pour ensuite calculer :

$$r = e^X \approx 2^k \cdot e^Y.$$

Cette réduction d'argument est réalisée en deux étapes : tout d'abord le calcul de k, obtenu comme un arrondi à l'entier le plus proche du produit en virgule fixe de X par $\frac{1}{\log 2}$, puis le calcul de Y comme la différence de X et de $k \cdot \log 2$.

Le calcul de e^Y reste très coûteux. On va donc appliquer une seconde réduction d'argument, en découpant Y en Y_1 et Y_2 :

$$Y = Y_1 + Y_2,$$

avec Y_1 les p bits de poids fort de la partie fractionnaire de Y et $0 \leq Y_2 < 2^{-p}$. On calcule alors :

$$e^Y = e^{Y_1} \times e^{Y_2},$$

en tabulant la valeur de e^{Y_1}.

Cette méthode est meilleure que la précédente pour deux raisons principales :

– le calcul de k ne nécessite pas une grande précision : la réduction d'argument sera précise dès lors que la soustraction $Y = X - k \log 2$ l'est. Ainsi, on va se contenter pour le calcul de k d'un petit multiplieur, et le calcul de Y ne nécessitera qu'un multiplieur rectangulaire pour calculer le produit du petit entier k par la constante $\log 2$. Cependant, comme k est cette fois-ci approché, l'exposant du résultat sera $k \pm 1$, ce qui signifie qu'une normalisation finale de la mantisse sera nécessaire ;

– le calcul de e^{Y_2} est ici bien plus simple que le calcul de $2^{F_{Y,2}}$ requis par l'algorithme précédent. En effet, comme Y_2 est très proche de 0, on a la formule de Taylor suivante :

$$e^{Y_2} = 1 + Y_2 + f(Y_2),$$

avec :

$$f(Y_2) = e^{Y_2} - 1 - Y_2 = \frac{1}{2} Y_2^2 + O(Y_2^3).$$

On peut ainsi vérifier que, comme $0 \leq Y_2 < 2^{-p}$, on a bien $0 \leq f(Y_2) < 2^{-2p}$, ce qui permet de simplifier grandement la table servant à évaluer cette fonction.

Enfin, la reconstruction est elle aussi relativement simple, puisque :

$$\begin{aligned} e^Y &= e^{Y_1} \times e^{Y_2} \\ &= e^{Y_1} \times (1 + Y_2 + f(Y_2)) \\ &= e^{Y_1} + e^{Y_1} \times (Y_2 + f(Y_2)), \end{aligned}$$

ce qui va nous permettre d'utiliser ici aussi un multiplieur rectangulaire de l'ordre de w_F bits par $w_F - p$ bits suivi d'une simple addition.

L'algorithme précédent utilise donc deux tables :

– la première, adressée par p bits, contient les valeurs de e^{Y_1} sur une grande précision (de l'ordre de w_F bits), et ne peut donc être compressée (Detrey *et al.*, 2005d) ;

– par contre, la seconde, adressée par les $w_F - 2p$ bits de poids fort de Y_2 et calculant la fonction $f(Y_2)$ sur autant de bits, peut pour sa part être optimisée grâce à ces méthodes à base de table. Nous utilisons donc la méthode HOTBM pour implémenter cette table.

Bien sûr, pour de plus faibles précisions, l'algorithme complet pour le calcul de e^Y peut être implémenté comme une seule table de valeur.

3.4. *Architecture*

L'architecture de cet opérateur est présentée figure 3. Il s'agit là aussi d'une implémentation directe de l'algorithme détaillé précédemment, qui peut être aisément pipelinée du fait de sa structure séquentielle.

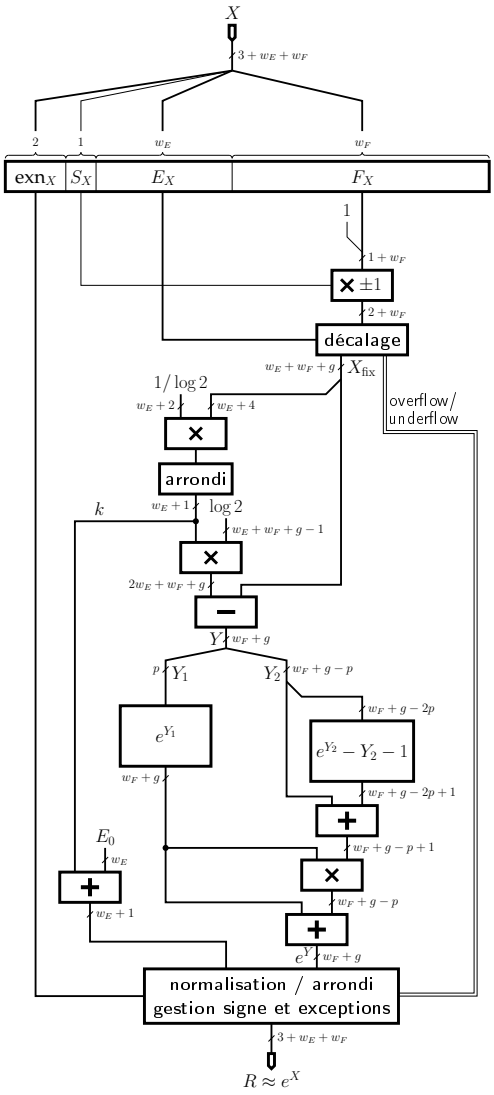


Figure 3. *Architecture de l'exponentielle*

Cette architecture a deux paramètres, p et g. Le premier, défini précédemment, permet de contrôler le compromis entre les tailles des deux tables de l'opérateur, et sa

valeur est généralement comprise entre $\frac{1}{4}w_F$ et $\frac{1}{3}w_F$. Le second désigne le nombre de bits de garde nécessaires pour garantir la précision du résultat R.

Quelques commentaires sur cette architecture :

– l'opérateur de décalage se charge de décaler la mantisse signée $\pm\overline{1,F_X}$ selon la valeur de l'exposant $E_X - E_0$ pour obtenir X_{fix}, la représentation de X en virgule fixe. Le décalage de la mantisse est d'au plus $w_E - 1$ bits vers la gauche (si le décalage dépasse cette valeur c'est qu'il s'agit d'un dépassement de capacité vers 0 ou vers $+\infty$) et d'au plus $w_F + g$ bits vers la droite (car lorsque X est proche de 0, e^X est proche de 1, donc seuls $w_F + g$ bits de précision suffisent). Le résultat du décalage est ensuite tronqué à ses $1 + w_E + w_F + g$ bits de poids fort, sa partie entière étant de $1 + w_E$ bits ;

– la valeur de k est choisie de sorte que la soustraction entre X_{fix} et $k \cdot \log 2$ annule la partie entière ainsi que le bit de poids fort de la partie fractionnaire de X. Ainsi, on s'assure bien que $-\frac{1}{2} < Y < \frac{1}{2}$;

– si la multiplication finale entre e^{Y_1} et e^{Y_2} est représentée ici comme un produit suivi d'une somme, selon l'architecture cible, une implémentation comme un unique multiplieur peut être plus intéressante (Beuchat *et al.*, 2002). De plus, comme il s'agit ici d'une multiplication par un ensemble de constantes, il est aussi possible de réaliser quelques optimisations sur cet opérateur (Boullis *et al.*, 2005) ;

– la normalisation finale peut avoir à décaler la mantisse du résultat d'au plus un bit vers la gauche.

Le lecteur intéressé par une analyse d'erreur détaillée, qui donne notamment la valeur de g, la trouvera dans la thèse de Jérémie Detrey (Detrey, 2007).

4. Les fonctions sinus et cosinus : implémentation de référence

Nous présentons ici une architecture d'évaluation du sinus et du cosinus comparable aux implémentations logicielles existantes : argument en radian, et arrondi fidèle, même pour de très grands arguments pour lesquels la pertinence d'une telle précision est discutable puisque le dernier bit de la mantisse peut peser plusieurs fois la période de la fonction.

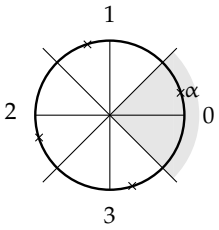
4.1. *Propriétés et réduction d'argument pour les angles en radian*

Il s'agit de ramener un nombre flottant x qui peut être très grand dans l'intervalle $\left[-\frac{\pi}{4}, \frac{\pi}{4}\right]$, en calculant un entier k et un réel α tels que

$$\alpha = x - k\frac{\pi}{2} \in \left[-\frac{\pi}{4}, \frac{\pi}{4}\right].$$

Une fois cette réduction d'argument effectuée, le sinus ou le cosinus du nombre de départ est déduit du sinus ou du cosinus de l'argument réduit modulo une identité trigonométrique triviale, donnée par la figure 4. Il faut y ajouter que l'on se ramène toujours à $\alpha \in [0, \pi/4]$ par simple changement de signe :

$$\begin{cases} \sin(-\alpha) = -\sin(\alpha), \\ \cos(-\alpha) = \quad \cos(\alpha). \end{cases}$$



0	$\begin{cases} \sin(x) = \sin(\alpha) \\ \cos(x) = \cos(\alpha) \end{cases}$	1	$\begin{cases} \sin(x) = \cos(\alpha) \\ \cos(x) = -\sin(\alpha) \end{cases}$
2	$\begin{cases} \sin(x) = -\sin(\alpha) \\ \cos(x) = -\cos(\alpha) \end{cases}$	3	$\begin{cases} \sin(x) = -\cos(\alpha) \\ \cos(x) = \sin(\alpha) \end{cases}$

Figure 4. *Réduction d'argument par quadrants*

Ici, k est défini comme l'arrondi de $x \times \frac{2}{\pi}$ à l'entier le plus proche. Nous allons utiliser une variante (présentée par Markstein (2000)) qui définit comme argument réduit y la partie fractionnaire de $x \times \frac{4}{\pi}$, et évalue ensuite $\sin\left(\frac{\pi}{4}y\right)$ et $\cos\left(\frac{\pi}{4}y\right)$. Cette variante présente deux avantages. D'une part, il y a une soustraction et une multiplication en moins par rapport au calcul de $\alpha = x - k\frac{\pi}{2}$. D'autre part, nous évaluons le sinus et le cosinus de l'argument réduit par une méthode à base de tables (Detrey *et al.*, 2005d), qui est plus facile à mettre en œuvre pour un argument réduit dans l'intervalle $\left[-\frac{1}{2}, \frac{1}{2}\right]$ que dans un intervalle comme $\left[-\frac{\pi}{4}, \frac{\pi}{4}\right]$. L'inconvénient principal de cette variante est que le développement limité de $sin\left(\frac{\pi}{4}y\right)$, dont on a besoin lors de l'évaluation proprement dite, est légèrement moins simple que celui de $sin(\alpha)$.

La question difficile est de calculer $x \times \frac{4}{\pi}$ avec une précision suffisante. Naturellement, on va utiliser un multiplieur par la constante $\frac{4}{\pi}$, mais pour de très grandes valeurs de x, on a besoin d'une grande précision de cette constante, puisqu'on ne veut garder que la partie fractionnaire du résultat. On peut toutefois remarquer qu'on n'a pas besoin de calculer entièrement la partie entière k de $x \times \frac{4}{\pi}$: on a besoin juste des trois bits de poids faible de k pour déterminer l'octant de l'argument réduit. Les bits de poids plus forts n'ont pas besoin d'être calculés, puisqu'ils correspondent à des multiples entiers de la période. Comme x est un nombre en virgule flottante, c'est sa mantisse qui sera multipliée par $\frac{4}{\pi}$, et son exposant peut être utilisé pour tronquer à gauche la constante et ne garder que les bits dont on a besoin pour obtenir trois bits entiers du produit. Cette idée est dûe à Payne et Hanek et a été popularisée par K.C. Ng (Ng, 1992).

Ensuite, il faut multiplier par la mantisse de x au moins $2w_F$ bits de $\frac{4}{\pi}$. En effet, les w_F bits de poids forts du produit ne seront pas valides puisque calculés au moyen d'une constante $\frac{4}{\pi}$ tronquée.

Il reste encore une subtilité. La partie fractionnaire y de $x \times \frac{4}{\pi}$ peut s'approcher extrêmement près de zéro pour certaines valeurs de x. Ainsi, il peut arriver que y commence par une longue suite de zéros. En virgule fixe ce ne serait pas un souci, mais si l'on veut calculer un sinus en virgule flottante dont tous les bits sont significatifs, il faut commencer par assurer que tous les bits de y le soient. Pour cela, il faut ajouter encore g_K bits de garde à la constante $\frac{4}{\pi}$. Un algorithme dû à Kahan et Douglas (Muller, 2005) permet de calculer, pour un domaine flottant donné, quelle est la plus petite valeur possible de y, et donc la valeur de g_K. On constate que g_K est de l'ordre de w_F. Finalement, la réduction d'argument trigonométrique nécessite :

– de stocker la constante $\frac{4}{\pi}$ sur un nombre de bits de l'ordre de $2^{w_E-1} + 3w_F$,

– du matériel pour extraire de l'ordre de $3w_F$ bits de cette constante,

– un multiplieur de l'ordre de $w_F \times 3w_F$ bits,

– un décaleur pour renormaliser éventuellement le résultat.

Le coût en matériel est donc élevé. Le coût en temps peut toutefois être réduit par une architecture à double chemin de calcul (ou *dual path*, comme on en trouve dans les additionneurs flottants), présentée ci-après.

D'autres réductions d'argument sont présentées dans (Muller, 2005). La technique de Cody et Waite n'est utile que pour les petits arguments dans une approche logicielle utilisant des opérateurs flottants. Les variantes de la réduction d'argument modulaire (Daumas *et al.*, 1995 ; Villalba *et al.*, 2006) ont été considérées, mais s'avèrent plus coûteuses que l'approche présentée ci-avant. Plus précisément, il s'agit d'approches itératives qui se prêtent très mal à une implémentation pipelinée.

4.2. *Architecture duale sinus et cosinus*

Si la réduction d'argument est coûteuse, il est possible de la partager dans le cas fréquent où l'on doit calculer le sinus et le cosinus d'un même angle (par exemple pour calculer une rotation). En fait, telle que présentée ci-avant, la réduction d'argument oblige l'opérateur à calculer toujours en parallèle le sinus et le cosinus de l'angle, puisque le résultat final sera l'un ou l'autre suivant le quadrant.

L'architecture générale de cette implémentation est donnée par la figure 5. L'architecture spécifique à la réduction d'argument, présentée plus en détail dans la section 4.2.1, est représentée figure 6. De même, les architectures détaillées pour le calcul du sinus et du cosinus sont données figures 7(a) et 7(b) respectivement.

L'étage de reconstruction quant à lui se contente d'appliquer les identités trigonométriques présentées figure 4 pour retrouver $\sin x$ et $\cos x$ à partir de $\sin\left(\frac{\pi}{4}y\right)$ et $\cos\left(\frac{\pi}{4}y\right)$, en fonction de l'octant indiqué par k. Le signe de x est aussi pris en compte dans le cas du sinus.

Enfin, une petite unité dédiée permet de traiter les cas exceptionnels, comme par exemple le calcul de $\sin(+\infty)$ qui renverra NaN.

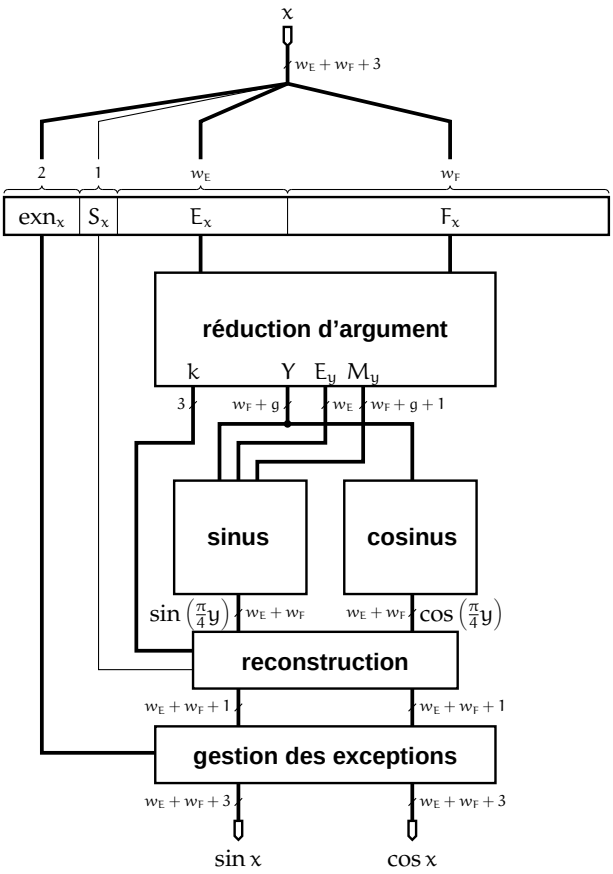


Figure 5. *Vue d'ensemble de l'architecture de l'opérateur dual sinus/cosinus*

4.2.1. *Réduction d'argument à double chemin*

Il faut décrire plus en détail la sortie de la réduction d'argument. On veut pouvoir calculer $\sin\left(\frac{\pi}{4}y\right)$ et $\cos\left(\frac{\pi}{4}y\right)$, tous deux en virgule flottante. Le cosinus sera compris entre $\frac{\sqrt{2}}{2}$ et 1, donc son exposant est connu. Par conséquent, pour le calculer, on peut se contenter d'une représentation de y en virgule fixe, notée Y. Par contre, le sinus peut s'approcher très près de zéro, donc il faut avoir également une représentation flottante de y, dénotée (E_y, M_y).

Dans le cas général, cette représentation flottante est obtenue par un compteur de zéros de poids forts (LZC) couplé à un décaleur. Mais lorsque x, le nombre de départ, est proche de zéro (en pratique $x < \frac{1}{2}$), on peut déduire directement l'exposant de y de celui de x, puisque les deux nombres sont dans un rapport constant de $\frac{4}{\pi}$ (il faut au plus une petite normalisation par un décalage de un bit). Donc on obtient M_y

et E_y rapidement, par contre, on obtient Y par un décalage variable de M_y suivant l'exposant E_y.

On obtient donc deux chemins de calcul exclusifs visibles sur la figure 6 : le chemin *close* pour les valeurs de x proches de zéro, qui calcule Y à partir de (E_y, M_y), et le chemin *far*, pour les valeurs de x distantes de zéro, qui calcule (E_y, M_y) à partir de Y.

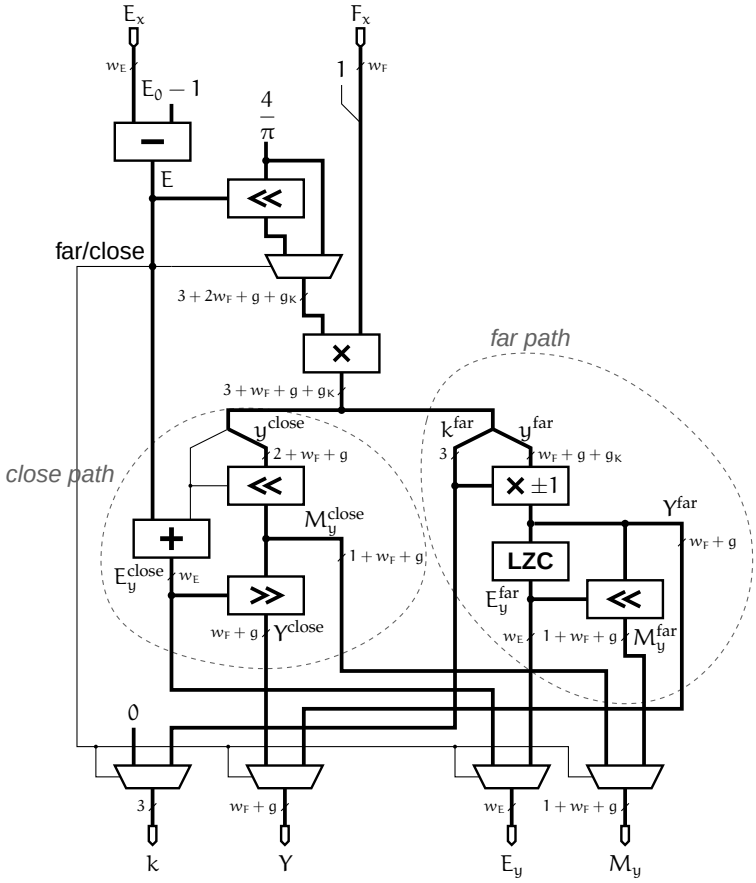


Figure 6. *Architecture détaillée de la réduction d'argument* dual path

4.2.2. *Evaluation par table de* $\cos\left(\frac{\pi}{4}y\right)$ *et* $\sin\left(\frac{\pi}{4}y\right)$

Pour le cosinus, on peut utiliser directement une table HOTBM puisque l'exposant du résultat est connu, autrement dit le résultat est en virgule fixe. On économise toutefois le stockage du premier bit, qui vaut toujours 1, en choisissant d'évaluer la fonction

$$f_{\cos}(y) = 1 - \cos\left(\frac{\pi}{4}y\right).$$

Le sinus, par contre, peut s'approcher près de zéro, et donc aura un exposant variable. Comme pour le logarithme, pour se ramener à une fonction en virgule fixe, on utilise l'équation

$$\sin\left(\frac{\pi}{4}y\right) = y \times \frac{\sin\left(\frac{\pi}{4}y\right)}{y}$$

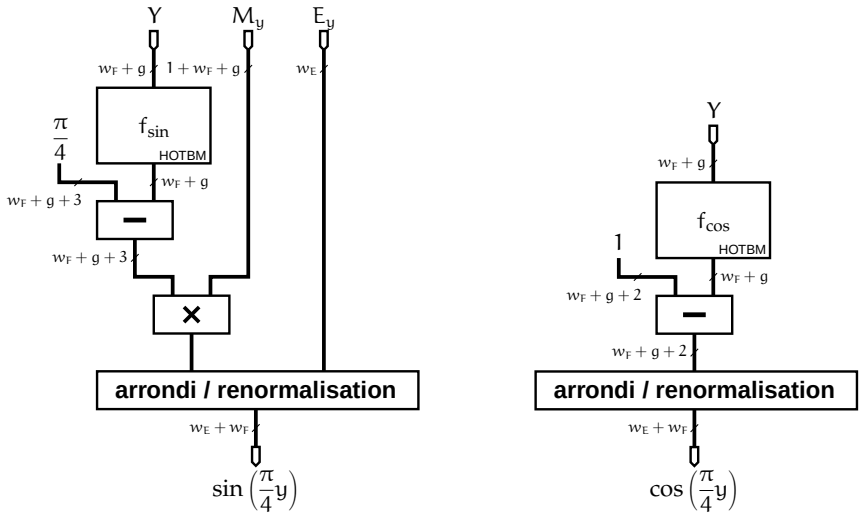
L'intérêt est que le terme de droite du produit peut être tabulé en virgule fixe : son développement de Taylor est

$$\frac{\sin\left(\frac{\pi}{4}y\right)}{y} \approx \frac{\pi}{4} + O(y^2),$$

alors que l'exposant du résultat est déterminé par l'exposant de y, E_y. On choisit donc de tabuler la fonction

$$f_{\sin}(y) = \frac{\pi}{4} - \frac{\sin\left(\frac{\pi}{4}y\right)}{y}.$$

La multiplication par y est une multiplication d'un nombre en virgule flottante (E_y, M_y) par un nombre en virgule fixe, et donc relativement simple.



(a) Sinus (b) Cosinus

Figure 7. *Architecture détaillée de l'évaluation du sinus (a) et du cosinus (b)*

4.3. *Analyse d'erreur*

Une analyse d'erreur similaire à celle présentée pour le logarithme montre que $g = 2$ bits de garde suffisent à garantir l'arrondi fidèle quelle que soit la précision.

5. Implémentations non standard des fonctions trigonométriques

Plusieurs alternatives à l'opérateur dual sinus/cosinus de référence permettent divers compromis entre précision, surface et latence. Les résultats des expériences réalisées (présentés dans la section suivante) montrent une économie de surface allant de 20 % à 50 %.

5.1. *Opérateur seul*

Certaines applications n'ont besoin de calculer que l'une des deux fonctions, sinus ou cosinus. Nous avons donc étudié une version de notre opérateur ne calculant que le sinus de x.

Pour éviter d'avoir à calculer le cosinus, la réduction d'argument doit se faire sur les quadrants et non les octants du cercle unité. Ainsi, l'angle réduit α sera dans l'intervalle $\left[0, \frac{\pi}{2}\right]$, deux fois plus large que pour l'opérateur dual.

La réduction est quasiment identique à la réduction d'argument de l'opérateur dual, si ce n'est la constante qui est $\frac{2}{\pi}$ au lieu de $\frac{4}{\pi}$. Cela ne va donc pas permettre d'économiser de matériel : le gros multiplieur $w_F \times 3w_F$ bits est toujours nécessaire.

Quant à l'évaluation de la fonction $f_{\sin}(y)$, puisqu'elle s'effectue sur un intervalle deux fois plus large, l'opérateur HOTBM va occuper plus de place. Au final, le gain de cet opérateur sur l'opérateur dual restera donc modéré.

5.2. *Une implémentation avec sortie en virgule fixe*

Une grande part de la surface et du délai de l'opérateur sont dus au multiplieur utilisé pour la réduction d'argument. En relâchant les contraintes de précision, on peut diminuer le nombre de bits de la constante $\frac{4}{\pi}$ à prendre en compte, et ainsi directement diminuer la taille et la latence de ce multiplieur.

Considérer un opérateur qui fournirait sa sortie en virgule fixe nous a semblé une option intéressante, puisqu'elle permettait justement d'éliminer les g_K bits de garde supplémentaires nécessaires pour la constante, mais aussi simplifiait le calcul de $\sin\left(\frac{\pi}{4}y\right)$. En effet, n'ayant plus besoin de renormaliser le résultat, on peut évaluer la fonction sinus directement à l'aide de HOTBM, sans passer par la fonction $f_{\sin}$. On économise donc alors aussi la multiplication par M_y.

5.3. *Fonctions trigonométriques en degrés et en* πx

On l'a vu, la difficulté de la réduction d'argument pour les fonctions en radian tient à ce que la constante $\frac{4}{\pi}$ est irrationnelle et qu'on a besoin d'un grand nombre de bits de cette constante. Dès lors que la constante est rationnelle et s'exprime sur peu de

bits (angles en degrés ou $sin(\pi x)$), le problème devient beaucoup plus simple. Par exemple, pour $sin(\pi x)$, la réduction d'argument consiste simplement à décomposer x en sa partie entière et sa partie fractionnaire. Comme x est un flottant, cela coûte tout de même un décalage, mais l'avantage est que cette opération est toujours exacte : la précision intermédiaire requise reste de l'ordre de w_F bits, pas plus. De plus, y étant calculé exactement, on a besoin d'un bit de garde de moins pour l'évaluation des fonctions $f_{\sin}$ et $f_{\cos}$.

6. Résultats

6.1. *Surface et délai*

Tous les opérateurs présentés dans cet article ont été synthétisés, placés et routés pour diverses précisions. Ces résultats ont été obtenus sous Xilinx ISE et XST 7.1, en ciblant un FPGA Virtex-II XC2V1000-4. Ces résultats sont présentés dans les tableaux 1 à 3, en termes de *slices* et de pourcentage de la surface du FPGA, et en termes de nanosecondes pour la latence des opérateurs.

Dans un souci de portabilité, les opérateurs ne nécessitent pas forcément l'utilisation des petits multiplieurs 18×18 bits présents dans les modèles récents de FPGA. Cependant, les opérateurs peuvent tirer parti de ces multiplieurs si ceux-ci sont disponibles. Nous présentons donc les résultats pour les deux alternatives : multiplieurs synthétisés sur la logique du FPGA, ou bien sur les petits multiplieurs dédiés.

Tous les FPGA actuels disposent aussi de petits blocs de mémoire embarqués. Cependant, du fait de la taille fixe (généralement 18k bits) de ces blocs et malgré la souplesse de leurs modes d'adressage, l'utilisation de ceux-ci est généralement sous-optimale et gâche trop de matériel. Par exemple, dans le cas du logarithme en simple précision, deux blocs de mémoire sont nécessaires au stockage des tables, soit 36k bits, alors que seuls 15k bits sont effectivement utilisés. De plus, l'accès à ces blocs demande un étage de pipeline supplémentaire. Pour plus de lisibilté, nous ne présentons pas dans cet article de résultats de placement/routage avec ces blocs mémoire. Cependant, leur utilisation reste possible en changeant une option du synthétiseur.

6.2. *Versions pipelinées des opérateurs*

Nos opérateurs sont aussi disponibles en version pipelinée. Le pipeline se traduit par un surcoût en surface d'environ 10 % par rapport à l'opérateur combinatoire. Le tableau 4 compare les performances des opérateurs simple précision $(w_E, w_F) = (8, 23)$ avec celles mesurées sur un processeur Intel Xeon cadencé à $2{,}4$ GHz utilisant les fonctions définies dans la `libc` GNU (reposant elle-même sur les instructions microcodées du processeur). Comme le montre ce tableau, malgré une fréquence d'horloge bien moindre, l'implémentation complètement pipelinée de l'opérateur sur FPGA permet d'atteindre des débits supérieurs à ceux du processeur généraliste.

Précision (w_E, w_F)	Multiplieurs	Surface (*slices*	%	mults)	Délai (ns)
$(3, 6)$	logique	129	(2%)	–	37
	18×18	101	(1%)	2	35
$(5, 10)$	logique	271	(5%)	–	49
	18×18	156	(3%)	3	46
$(6, 13)$	logique	439	(8%)	–	57
	18×18	247	(4%)	3	50
$(7, 16)$	logique	619	(12%)	–	69
	18×18	355	(6%)	6	68
$(8, 23)$	logique	1447	(27%)	–	86
	18×18	876	(16%)	9	76

Tableau 1. *Surface et délai de l'opérateur pour le logarithme*

Précision (w_E, w_F)	Multiplieurs	Surface (*slices*	%	mults)	Délai (ns)
$(3, 6)$	logique	143	(2%)	–	55
	18×18	67	(1%)	3	48
$(5, 10)$	logique	269	(5%)	–	72
	18×18	130	(2%)	4	62
$(6, 13)$	logique	364	(7%)	–	78
	18×18	188	(3%)	5	73
$(7, 16)$	logique	500	(9%)	–	80
	18×18	269	(5%)	5	81
$(8, 23)$	logique	938	(18%)	–	97
	18×18	512	(10%)	9	96

Tableau 2. *Surface et délai de l'opérateur pour l'exponentielle*

Il convient cependant de tempérer ces résultats par quelques remarques. Tout d'abord, nos opérateurs sont sensiblement moins précis que l'algorithme logiciel, qui calcule avec une précision interne de 80 bits, ce qui lui assure presque certainement l'arrondi correct du résultat pour toutes les entrées. De plus, des jeux d'instructions plus récents offrent des latences moindres pour les fonctions élémentaires. Ainsi, le processeur Intel Itanium 2 peut calculer un logarithme simple précision en 32 cycles puis fournir un résultat presque tous les 5 cycles (Thomas *et al.*, 2004), soit un débit de près de $300 \cdot 10^6$ calculs par seconde pour un processeur cadencé à $1{,}5$ GHz. On peut cependant toujours mettre plusieurs opérateurs en parallèle sur le FPGA pour augmenter son débit.

	Dual sinus/cosinus				Sinus seul			
Précision	Surface			Latence	Surface			Latence
(w_E, w_F)	(*slices*	%	mults)	(ns)	(*slices*	%	mults)	(ns)
$(5, 10)$	803	(15%)	–	69	709	(13%)	–	70
	424	(8%)	7	61	310	(6%)	6	62
$(6, 13)$	1159	(22%)	–	86	1027	(20%)	–	86
	537	(10%)	7	76	474	(9%)	6	79
$(7, 16)$	1652	(32%)	–	91	1428	(27%)	–	92
	816	(15%)	10	87	609	(11%)	10	83
$(7, 20)$	2549	(49%)	–	99	2050	(40%)	–	101
	1372	(26%)	17	96	979	(19%)	15	92
$(8, 23)$	3320	(64%)	–	109	2659	(51%)	–	105
	1700	(33%)	19	99	1279	(24%)	16	100

	Sinus/cosinus en virgule fixe				Sinus/cosinus en πx			
Précision	Surface			Latence	Surface			Latence
(w_E, w_F)	(*slices*	%	mults)	(ns)	(*slices*	%	mults)	(ns)
$(5, 10)$	376	(7%)	–	57	363	(7%)	–	58
	241	(4%)	4	51	244	(4%)	3	46
$(6, 13)$	642	(12%)	–	63	641	(12%)	–	61
	380	(7%)	4	58	462	(9%)	3	61
$(7, 16)$	923	(18%)	–	72	865	(16%)	–	73
	528	(10%)	5	70	559	(10%)	4	69
$(7, 20)$	1620	(31%)	–	82	1531	(29%)	–	84
	973	(19%)	9	75	1005	(19%)	8	76
$(8, 23)$	2203	(43%)	–	88	2081	(40%)	–	89
	1294	(25%)	12	80	1365	(25%)	10	85

Tableau 3. *Surface et délai des opérateurs trigonométriques*

Fonction	Cible	Cycles	Délai (ns)	Débit (10^6 op/s)
log	Intel Xeon à 2,4 GHz	196	82	12
	FPGA Virtex-II à 100 MHz	11	86	100
exp	Intel Xeon à 2,4 GHz	308	128	8
	FPGA Virtex-II à 100 MHz	15	97	100
sin / cos	Intel Xeon à 2,4 GHz	206	86	12
	FPGA Virtex-II à 100 MHz	18	109	100

Tableau 4. *Performances comparées du FPGA et du processeur Intel Xeon, en simple précision. Le délai donné pour le FPGA est le délai combinatoire*

Il faut aussi comparer nos travaux à ceux de Doss *et al.* (2004), qui ont décrit un opérateur pour le calcul de l'exponentielle en simple précision. Leur algorithme est inspiré du logiciel, et effectue tous ses calculs intermédiaires en virgule flottante. Leur opérateur occupe 5564 *slices* et calcule son résultat en 74 cycles à 85 MHz sur un FPGA Virtex-II XC2V4000-5 (légèrement supérieur à celui utilisé pour nos estimations de performance). Notre approche est donc à la fois cinq fois plus compacte et cinq fois plus rapide.

7. Vers la double précision

Comme la littérature récente (deLorimier *et al.*, 2005 ; Dou *et al.*, 2005) s'intéresse de plus en plus au calcul en double précision sur FPGA ($w_E = 11$, $w_F = 52$), nous comptons adapter FPLibrary pour qu'elle passe à l'échelle pour ce format. C'est déjà le cas pour les quatre opérations, même si des optimisations à la marge sont possibles. Pour les fonctions élémentaires, par contre, les méthodes à base de tables montrent leurs limites aux alentours de 32 bits (Detrey *et al.*, 2005d). Le problème est que la taille des tables utilisées par la méthode HOTBM augmente de manière exponentielle avec la précision w_F. Cette croissance s'observe dans les tables précédentes.

Pour atteindre la double précision, il faut pousser plus loin la réduction d'argument. Pour l'exponentielle, il s'agit de réitérer la seconde réduction d'argument : au lieu de découper Y en deux sous-mots Y_1 et Y_2, on peut le découper en trois, quatre ou plus sous-mots Y_i. Cela augmente le nombre de multiplieurs et le nombre de tables.

Le dernier Y_i restera de taille environ $w_F/2$, car alors son exponentielle pourra être évaluée précisément et rapidement par le développement de Taylor à l'ordre 2.

Pour choisir le découpage en sous-mots de la première moitié de Y, on peut tirer parti de la structure des FPGA à base de petites tables à 4 entrées : la surface totale de tables sera minimale si on décompose la première moitié de Y en sous-mots de 4 bits, donc au nombre d'environ $w_F/8$. Pour la reconstruction, on aura donc le même nombre de multiplieurs.

Pour minimiser la taille des multiplieurs, une idée couramment utilisée (voir par exemple (Ercegovac, 1973), (Wrathall *et al.*, 1978), (Wong *et al.*, 1994) et le chapitre sur Cordic/BKM de (Muller, 2005)) est de ne pas tabuler e^{Y_i} mais un nombre proche P_i qui s'écrit sur peu de bits. Mais alors il faut compenser l'approximation ainsi faite : Y_{i+1} n'est plus simplement composé des bits suivants de Y, il faut le corriger par une valeur tabulée de $\log P_i$. Cette correction est une addition qui doit être précise, mais coûte moins qu'une multiplication précise.

La réduction d'argument devient ainsi itérative. On peut obtenir ainsi une architecture dont la surface est quadratique en la précision, et non plus exponentielle, puisque la somme des tailles des multiplieurs correspond *grosso modo* à un multiplieur complet de taille $w_F \times w_F$. Il en est de même de la somme des tailles de tables et des additionneurs.

Une variante de cette réduction d'argument itérative permet de calculer le logarithme. Nos premières expériences (Detrey *et al.*, 2007) montrent qu'on peut ainsi obtenir une exponentielle ou un logarithme en double précision entre deux et trois fois plus gros que les opérateurs simple-précision présentés ici. Leur latence, toutefois, est bien plus élevée.

Enfin, une telle réduction itérative peut aussi permettre de calculer les fonctions trigonométriques : pour ces dernières, l'identité complexe $e^{ix} = \cos(x) + i\sin(x)$ se traduit par des formules comme $\sin(Y_1 + Y_2) = \sin(Y_1)\cos(Y_2) + \cos(Y_1)\sin(Y_2)$, analogue de $e^{Y_1+Y_2} = e^{Y_1}e^{Y_2}$. De fait, nous avons considéré l'utilisation de telles identités pour nos opérateurs trigonométriques, mais la complexité accrue de l'opérateur obtenu ne se justifiait pas pour les précisions inférieures à la simple précision. Une étude fine en visant les précisions plus élevées et en tirant parti des ressources du FPGA reste à faire.

8. Conclusion et perspectives

Cet article décrit des opérateurs matériels pour le calcul des fonctions logarithme, exponentielle, sinus et cosinus en virgule flottante paramétrée sur les FPGAs. Ces opérateurs se caractérisent par une grande qualité numérique et par l'utilisation d'algorithmes ciblés pour les FPGAs offrant des performances élevées en consommant peu de ressources. Malgré une fréquence de fonctionnement vingt fois plus faible, leur débit de calcul surpasse celui des processeurs généralistes d'un facteur dix. Ces opérateurs permettent de calculer en simple précision, et des pistes sont données pour atteindre la double précision.

En raison du coût de la réduction d'argument trigonométrique, plusieurs versions non standard de sinus et cosinus ont été également décrites, en espérant que les retours d'utilisateurs diront lesquels de ces opérateurs sont les plus utiles, et dans quel contexte.

L'objectif principal de nos prochains travaux est de passer à la double précision. Toutefois, même pour les petites précisions, il reste toujours des optimisations possibles. Par exemple, chaque architecture utilise des multiplieurs par des constantes. Pour ces opérateurs, il est possible d'utiliser diverses techniques d'optimisations (Chapman, 1994 ; de Dinechin *et al.*, 2000).

Une question plus générale est la méthodologie de conception de tels opérateurs. Pour l'instant, une partie du VHDL est écrite à la main, et une autre (par exemple les tables HOTBM) est produite par des générateurs écrits en C++. Nous cherchons à aller vers plus d'automatisation, notamment pour ce qui est du calcul d'erreur et de la détermination des paramètres comme les nombres de bits de garde. A terme, les mêmes outils pourraient aider à porter sur FPGA des calculs flottants arbitraires.

9. Bibliographie

Anderson C. S., Story S., Astafiev N., "Accurate math functions on the Intel IA-32 architecture : A performance-driven design", *7th Conference on Real Numbers and Computers*, Nancy, France, July, 2006, p. 93-105.

Beuchat J.-L., Tisserand A., "Small multiplier-based multiplication and division operators for Virtex-II devices", M. Glesner, P. Zipf, M. Renovell (eds), *Field-Programmable Logic and Applications : 12th International Conference, FPL 2002*, n° 2438 in *Lecture Notes in Computer Science*, Springer-Verlag, Montpellier, France, September, 2002, p. 513-522.

Boullis N., Tisserand A., "Some optimizations of hardware multiplication by constant matrices", *IEEE Transactions on Computers*, vol. 54, n° 10, October, 2005, p. 1271-1282.

Chapman K. D., "Fast integer multipliers fit in FPGAs", *EDN*, May, 1994.

Daumas M., Mazenc C., Merrheim X., Muller J. M., "Modular Range Reduction : A New Algorithm for Fast and Accurate Computation of the Elementary Functions", *Journal of Universal Computer Science*, vol. 1, n° 3, March, 1995, p. 162-175.

de Dinechin F., Lefèvre V., "Constant multipliers for FPGAs", H. Arabnia (ed.), *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00)*, vol. 1, CSREA Press, Las Vegas, NV, USA, June, 2000, p. 167-173.

deLorimier M., DeHon A., "Floating-point sparse matrix-vector multiply for FPGAs", *ACM/SIGDA Field-Programmable Gate Arrays*, ACM Press, 2005, p. 75-85.

Detrey J., Arithmétiques réelles sur FPGA : Virgule fixe, virgule flottante et système logarithmique, PhD thesis, École Normale Supérieure de Lyon, Lyon, France, January, 2007.

Detrey J., de Dinechin F., « Outils pour une comparaison sans *a priori* entre arithmétique logarithmique et arithmétique flottante », M. Auguin, D. Lavenier (eds), *Architecture des Ordinateurs*, vol. 24, n° 6 of *Technique et Science Informatiques*, Lavoisier, Cachan, France, November, 2005a, p. 625-643.

Detrey J., de Dinechin F., "A parameterizable floating-point logarithm operator for FPGAs", *39th Asilomar Conference on Signals, Systems & Computers*, IEEE Signal Processing Society, Pacific Grove, CA, USA, November, 2005b, p. 1186-1190.

Detrey J., de Dinechin F., "A parameterized floating-point exponential function for FPGAs", G. Brebner, S. Chakraborty, W.-F. Wong (eds), *IEEE International Conference on Field-Programmable Technology (FPT'05)*, IEEE, Singapore, December, 2005c, p. 27-34.

Detrey J., de Dinechin F., "Table-based polynomials for fast hardware function evaluation", S. Vassiliadis, N. Dimopoulos, S. Rajopadhye (eds), *16th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'05)*, IEEE Computer Society, Samos, Greece, July, 2005d, p. 328-333.

Detrey J., de Dinechin F., « Opérateurs trigonométriques en virgule flottante sur FPGA », *RenPar'17, SympA'2006, CFSE'5 et JC'2006*, Perpignan, France, October, 2006, p. 96-105.

Detrey J., de Dinechin F., Pujol X., "Return of the hardware floating-point elementary function", *Proceedings of the 18th IEEE Symposium on Computer Arithmetic (ARITH'18)*, IEEE Computer Society, Montpellier, France, June, 2007.

Doss C., Riley Jr. R. L., "FPGA-based implementation of a robust IEEE-754 exponential unit", J. Arnold, K. Pocek (eds), *12th Annual IEEE Symposium on Field-Programmable Custom*

Computing Machines (FCCM'04), IEEE Computer Society, Napa, CA, USA, April, 2004, p. 229-238.

Dou Y., Vassiliadis S., Kuzmanov G. K., Gaydadjiev G. N., "64-bit floating-point FPGA matrix multiplication", *ACM/SIGDA Field-Programmable Gate Arrays*, ACM Press, 2005.

Ercegovac M. D., "Radix-16 evaluation of certain elementary functions", *IEEE Transactions on Computers*, vol. 22, n° 6, June, 1973, p. 561-566.

Govindu G., Zhuo L., Choi S., Prasanna V., "Analysis of High-Performance Floating-Point Arithmetic on FPGAs", *Reconfigurable Architecture Workshop, Intl. Parallel and Distributed Processing Symposium*, IEEE Computer Society, 2004.

ISO/IEC, *International Standard ISO/IEC 9899 :1999(E). Programming languages – C*, 1999.

Lefèvre V., Muller J.-M., Tisserand A., "Toward correctly rounded transcendentals", *IEEE Transactions on Computers*, vol. 47, n° 11, November, 1998, p. 1235-1243.

Lienhart G., Kugel A., Männer R., "Using Floating-Point Arithmetic on FPGAs to Accelerate Scientific N-Body Simulations", *FPGAs for Custom Computing Machines*, IEEE, 2002.

Ligon W., McMillan S., Monn G., Schoonover K., Stivers F., Underwood K., "A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs", *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, USA, 1998.

Markstein P., *IA-64 and Elementary Functions : Speed and Precision*, Hewlett-Packard Professional Books, Prentice Hall, 2000.

Muller J.-M., *Elementary Functions, Algorithms and Implementation*, Birkhäuser, Boston, MA, USA, October, 2005.

Ng K. C., "Argument Reduction for Huge Arguments : Good to the Last Bit", *SunPro*, July, 1992.

Ortiz F. E., Humphrey J. R., Durbano J. P., Prather D. W., "A study on the design of floating-point functions in FPGAs", P. Cheung, G. Constantinides, J. de Sousa (eds), *Field-Programmable Logic and Applications : 13th International Conference, FPL 2003*, n° 2778 in *Lecture Notes in Computer Science*, Springer-Verlag, Lisbon, Portugal, September, 2003, p. 1131-1134.

Paul G., Wilson M. W., "Should the Elementary Functions Be Incorporated Into Computer Instruction Sets ?", *ACM Transactions on Mathematical Software*, vol. 2, n° 2, June, 1976, p. 132-142.

Tang P. P., "Table-lookup algorithms for elementary functions and their error analysis", P. Kornerup, D. Matula (eds), *Proceedings of the 10th IEEE Symposium on Computer Arithmetic (ARITH'10)*, IEEE Computer Society, Grenoble, France, June, 1991, p. 232-236.

Thomas J. W., Okada J. P., Markstein P., Li R.-C., The *libm* library and floating-point arithmetic in HP-UX for Itanium-based systems, Technical report, Hewlett-Packard Company, Palo Alto, CA, USA, December, 2004.

Underwood K., "FPGAs vs. CPUs : Trends in Peak Floating-Point Performance", *ACM/SIGDA Field-Programmable Gate Arrays*, ACM Press, 2004.

Villalba J., Lang T., Gonzalez M. A., "Double-Residue Modular Range Reduction for Floating-Point Hardware Implementations", *IEEE Transactions on Computers*, vol. 55, n° 3, March, 2006, p. 254-267.

Wong W. F., Goto E., “Fast Hardware-Based Algorithms for Elementary Function Computations Using Rectangular Multipliers”, *IEEE Transactions on Computers*, vol. 43, n° 3, March, 1994, p. 278-294.

Wrathall C., Chen T. C., “Convergence guarantee and improvements for a hardware exponential and logarithm evaluation scheme”, *Fourth IEEE Symposium on Computer Arithmetic*, October, 1978, p. 175-182.

Article reçu le 7 mars 2007
Accepté le 21 juin 2007

Jérémie Detrey *a obtenu son doctorat en informatique à l’Ecole Normale Supérieure de Lyon en janvier 2007. Ses travaux durant cette thèse ont porté sur l’arithmétique des ordinateurs, et plus particulièrement l’implantation matérielle d’opérateurs pour l’arithmétique réelle sur architectures reconfigurables.*

Florent de Dinechin *a obtenu son doctorat en informatique à l’Université de Rennes 1 en janvier 1997 puis son habilitation à diriger les recherches à l’Ecole Normale Supérieure de Lyon en juin 2007. Il s’intéresse à l’arithmétique des ordinateurs et plus précisément l’arithmétique flottante, tant du point de vue logiciel que matériel.*